KEKER & VAN NEST LLP
ROBERT A. VAN NEST - #84065
rvannest@kvn.com
CHRISTA M. ANDERSON - #184325
canderson@kvn.com
MICHAEL S. KWUN - #198945
mkwun@kvn.com
633 Battery Street
San Francisco, CA 94111-1809
Tel: 415.391.5400
Fax: 415.397.7188

KING & SPALDING LLP
DONALD F. ZIMMER, JR. - #112279
fzimmer@kslaw.com
CHERYL A. SABNIS - #224323
csabnis@kslaw.com
101 Second Street, Suite 2300
San Francisco, CA 94105
Tel: 415.318.1200
Fax: 415.318.1300

KING & SPALDING LLP
SCOTT T. WEINGAERTNER
(*Pro Hac Vice*)
sweingaertner@kslaw.com
ROBERT F. PERRY
rperry@kslaw.com
BRUCE W. BABER *(Pro Hac Vice)*
1185 Avenue of the Americas
New York, NY 10036
Tel: 212.556.2100
Fax: 212.556.2222

IAN C. BALLON - #141819
ballon@gtlaw.com
HEATHER MEEKER - #172148
meekerh@gtlaw.com
GREENBERG TRAURIG, LLP
1900 University Avenue
East Palo Alto, CA 94303
Tel: 650.328.8500
Fax: 650.328.8508

Attorneys for Defendant
GOOGLE INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.,

Plaintiff,

v.

GOOGLE INC.,

Defendant.

Case No. 3:10-cv-03561 WHA

GOOGLE'S MAY 23, 2012 COPYRIGHT LIABILITY TRIAL BRIEF

Dept.: Courtroom 8, 19th Floor
Judge: Hon. William Alsup

Google hereby responds to the Court's request for more briefing regarding interfaces, exceptions and interoperability. *See* Dkt. 1181. Like the other aspects of the SSO of the 37 API packages, interfaces and exceptions are functional requirements for compatibility with the APIs in those packages, and therefore are not copyrightable. *Sega Enters. Ltd. v. Accolade, Inc.*, 977 F.2d 1510, 1522 (9th Cir. 1992) (citing 17 U.S.C. § 102(b)). By implementing the SSO of the 37 API packages, Google increased the extent to which source code is compatible with both the Android and J2SE platforms.

I. The interfaces and exceptions that are publicly declared in the 37 API packages are functional requirements for compatibility with the APIs in those packages, and therefore are not copyrightable.

A. With respect to the 37 API packages, J2SE and Android declare substantially the same number of interfaces, and throw exactly the same number of exceptions.

In J2SE 5.0, the 37 API packages at issue include declarations for 171 interfaces, while in Android 2.2 ("Froyo"), the 37 API packages include declarations for 158 interfaces. These 158 interfaces in Android 2.2 are a subset of the 171 interfaces in J2SE 5.0, i.e., source code referencing, implementing or extending these 158 interfaces in Android 2.2 will also be compatible with J2SE 5.0. Exhibit A, attached hereto, shows the number of interface declarations on a package-by-package basis.[1] For most of the 37 packages, the number of interface declarations is the same in J2SE and Android. *See* Ex. A.

In both J2SE 5.0 and Android 2.2, the public methods in the 37 API packages throw 1,257 exceptions. Exhibit B, attached hereto, shows the number of exceptions thrown by the public methods on a package-by-package basis. For each of the 37 packages, the number of exceptions thrown is the same in J2SE and Android. *See* Ex. B.

B. Example: the Comparable interface.

Interfaces are a listing of methods and fields that "capture what is common across . . . very different things," with the purpose of allowing standardized, simplified interaction with those

[1] Exhibits A and B were both created by programmatic analysis of compiled versions of J2SE 5.0 and Android 2.2—i.e., programmatic analysis of compiled versions of the source code that was admitted into evidence as TX 623 (J2SE 5.0) and TX 46 (Android 2.2).

common features. RT 590:9-11 (Reinhold). Like classes and methods, interfaces have a declaration, and are part of the APIs at issue. TX 984 (*The Java Language Specification*, 3d ed.) at 114; RT 590:1-3 (Reinhold) ("The term Application Programming Interface includes these interfaces in the classes and methods and everything else.").

For example, the java.lang package includes a declaration for the "Comparable" interface. This interface has a single method, called compareTo. The source code declaration of the Comparable interface, without comments, is:

```
public interface Comparable<T> {
        . . .
        public int compareTo(T o);
}
```

Ex. C (excerpt from TX 623),[2] lines 82, 121-22. This means that any class that "implements" the Comparable interface must declare a method called "compareTo" that returns an integer and accepts a single argument that has the same "type" as the class being declared. The documentation for the compareTo method provides that if there are two objects called "x" and "y" that are instantiated from the same class, and that class implements the Comparable interface, the source code expression "x.compareTo(y)" will return a negative integer if x is less than y, a zero if x and y are equal, and a positive integer if x is greater than y. *See* Ex. C, lines 114-16. The Comparable interface includes only one method, but an interface can have additional methods or fields.

The Comparable interface allows a developer to declare a method that relies on the presence of the compareTo method that is promised for all classes that implement the Comparable interface. For example, the ComparableTimSort.java file, written by Josh Bloch, includes a method that sorts arrays of objects that implement the Comparable interface. *See* Ex. D (TX 45.2), lines 20-22. At various points in ComparableTimSort.java, the source code generically refers to a "Comparable" object, e.g.:

```
if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) {
```

[2] Exhibit C is a printed version of the file licenseebundles/source-bundles/tmp/j2se/src/share/classes/java/lang/Comparable.java, from TX 623.

Ex. D, line 286. The "(Comparable)" syntax indicates that the object "a" must be an object instantiated from a class that implements the Comparable interface. By making use of the interface construct, Josh Bloch was able to write the ComparableTimSort method in a manner that works for *any* array of Comparable objects. Indeed, if tomorrow a developer were to create a new class that implemented the Comparable interface, Josh Bloch's ComparableTimSort method would sort an array of objects instantiated from that new class, even though Josh Bloch could not have known about that developer's new class when he wrote the source code for the ComparableTimSort method.

Had Google not implemented the publicly declared interfaces that are in the 37 API packages, code that depends on them would not work. For example, if Android did not declare the Comparable interface, then a class that includes "implements Comparable" as part of its declaration would not compile. Moreover, had Android omitted the Comparable interface, methods that depend on it, like ComparableTimSort, would not function on the Android platform. Thus, because the public interfaces in the 37 API packages are functionally required for compatibility with the APIs in those packages, those interface declarations are not copyrightable. *Sega*, 977 F.2d at 1522 (citing 17 U.S.C. § 102(b)).

C. Example: the FileNotFoundException exception.

Exceptions are a type of class used by the Java language to communicate to a program that a particular error has occurred. TX 984 at 297. An exception can signal a problem internal to the program, such as having a beginning index that is greater than the ending index when sorting an array. An exception can also signal an external problem, such as a missing file.

When an error occurs, the method in which the error occurs is said to "throw" the relevant exception. *Id.* The method can then either address ("catch") the exception itself, or pass the exception on to the code that called the method.[3] In the latter case, the declaration of the method generally must include the word "throws" followed by the type of exception that is thrown. *Id.* at

[3] Because a method that throws an exception passes that exception to the code that calls the method, the exception can be thought of as part of the "input-output" schema for a method, although it is not the same as the "return" for the method.

394 (discussing inclusion of throw in method and constructor declarations); *see also id.* at 301-02, 222 (explaining why not all types of exceptions must be listed in the method declaration). Methods may throw more than one exception.[4] *Id.* at 221.

As an example, the method java.io.FileReader FileReader(File file) can throw the java.io.FileNotFoundException. This informs the code that called the FileReader method that the requested file could not be found, and thus could not be read. The declaration of the java.io.InputStream method indicates the type of exception that may be thrown so that developers invoking FileReader know that their code needs to "catch" that type of exception:

public FileReader(File file) *throws FileNotFoundException*

Ex. E (excerpt from TX 610.2[5]) (emphasis added).

The exceptions named in the throws clause are "part of the contract between the implementor [of the method] and the user [of the method—i.e., the developer writing source code that invokes the method]." TX 984 at 299. Because of this, the Java language specification requires the compiler to check to ensure that exceptions are properly handled. *Id.* at 299. For example, if "throws FileNotFoundException" is removed from the declaration of a method that throws that exception in the implementation of that method (such as the FileReader method), the class that contains the method will not compile. *Id.* at 301. In addition, if an application catches that exception, but the method that throws the exception does not have a throw clause with that exception in its declaration, the application that calls the method will fail to compile. *Id.*

As a result, maintaining the correct information about thrown exceptions in the method declaration is necessary for compatibility. Indeed, the thrown exceptions are *part* of the method declaration. *See* TX 984 at 210 (defining "*Throws*$_{opt}$" as part of the "*MethodHeader*"), 221 (discussing the "throws" clause). Because the thrown exceptions are part of the functional requirements for compatibility with the 37 API packages, they are not copyrightable. *Sega*, 977

[4] The table in Ex. B counts the total number of exceptions thrown by methods in the public API. Because more than one exception may be thrown by a method, this is larger than the number of methods that throw exceptions.

[5] Exhibit D is a printed version of the file /java/lang/Comparable.html, from TX 610.2. This file is also available on the web at http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Comparable.html.

F.2d at 1522 (citing 17 U.S.C. § 102(b)).

II. Because the SSO of the 37 API packages is functionally required for compatibility with the APIs in those packages, it is not copyrightable.

A. Because Android implements the SSO of the 37 API packages, code written using the APIs in those packages is interoperable between Android and J2SE.

There is no quantitative data in the trial record that demonstrates the extent to which J2SE applications written before Android was released are able to run on the Android platform. Nor is there any quantitative data in the trial record that demonstrates the extent to which post-Android applications are able to run both on the Android and J2SE platforms.

The record does demonstrate, however, that code that relies on APIs that are common to the two platforms will compile and execute on both platforms. RT 2172:6-11 (Astrachan) ("Q. Do you have an opinion, professor, whether, from a computer science perspective, Android and Java are compatible with respect to the methods and other constructors and other items in the classes of the 37 accused packages? A. Yes. For those 37 packages, the code that I write on one platform will run on the other platform."); *see also* RT 2171:24-2172:11 (Astrachan); RT 2287:1-8 (Mitchell) ("I think the point that was illustrated by this code and Dr. Astrachan's description of it is that, for a given piece of code such as this class that he wrote with a marker, it may run on both platforms if the only things it requires are things that are common to the two"); RT 2292:25-2293:14 (Mitchell) (agreeing that Dr. Astrachan's code would work both on the Android and J2SE platforms, and that calling the two platforms "compatible" in this sense is using "a great definition of 'compatible'"). Professor Astrachan explained this during trial as follows:

> Q. Have you formed an opinion, Professor, regarding what, if anything, accounts for the fact that the 37 packages in both have the same structure, organization, and use the same names?
>
> A. Those same names that we have in Android and in Java are needed so that the code inter-operates, so that code I write can be reused in another situation. So for the functionality of using those APIs, the method signatures need to be the same so that the code will inter-operate and meet programmer expectations.

RT 2183:2-11. Ensuring that the signatures used by Android for the APIs in the 37 packages match the signatures used in J2SE "is what allows me to use the libraries on both—use the code I write, like that code up there, on both platforms. Because I'm using those method signatures, my

code will function the same on both platforms." RT 2183:17-20; *see also* RT 2185:5-9 ("that structure of the names of the classes, packages, and methods needs to be the same so that the code will work on both platforms, be compatible, inter-operate, so that I can call the methods. Those need to be the same.").

Moreover, even to the extent that a J2SE application relies on J2SE APIs that are not supported on the Android platform—or to the extent that an Android application relies on Android APIs that are not supported on the J2SE platform—the portion of the source code that relies on APIs that are common to the two platforms will not need to be rewritten. Thus, even for applications that rely on APIs that are not common to both platforms, the Android and J2SE platforms are still *partially* compatible. Indeed, Professor Mitchell testified that one reason why he believes Google wanted to use the APIs in the 37 packages is because those APIs "are known *and used in existing code*." RT 2289:21-13 (emphasis added); *see also* RT 1787:23-1788:4 (Bornstein) ("And, actually, not even just a matter of comfort, but there's a lot of source code out there that wasn't—you know, wasn't written by—well, that was written by lots of people that already existed that could potentially work just fine on Android. And if we went and changed all the names of things, then that source code wouldn't just work—").

By implementing these core APIs, Google reduced the effort required to "port" an application from one platform (e.g., the J2SE platform) to another (e.g., the Android platform), thus promoting increased interoperability. This effort is similar to what is necessary to "port" applications from, for example, the J2SE platform to a given profile of the J2ME platform, or from one profile of the J2ME platform to another. As Dr. Reinhold testified:

> Write once, run anywhere was never a promise that if you wrote code for one Java platform that it would automatically/magically work on another.
>
> The write once, run anywhere promise is relative to a one of the Java platforms. If you write an application that uses Java SE 5, then you can run it on Sun's implementation, on Oracle's implementation, on IBM's implementation, and on others.
>
> Will that same code run on a particular configuration of Java ME? Well, it depends. *It might. It might not. It depends which APIs it uses.*

RT 725:10-20 (emphasis added).

In addition, because the Android platform shares a common core set of APIs with the J2SE platform, developers are able to use experience they gain from working with one platform when developing applications for the other platform. Developers expect these core APIs when they write code in the Java language. RT 2202:6-11, 2203:11-15 (Astrachan); RT 2291:1-8 (Mitchell); RT 364:17-21 (Kurian); RT 519:16-520:6 (Screven). In this sense, Android is compatible with the skills and expectations of Java language programmers.

Finally, the record establishes that interoperability was a motive of Google at the time it made the decision to implement the 37 API packages. Google chose the 37 API packages precisely *because* Java language developers expect them to be present when they write code in the Java language. RT 1782:6-1783:10 (Bornstein). "The goal of the project was to provide something that was familiar to developers." RT 1783:19-21. And in hiring the contractor Noser to help write source code implementing the 37 API packages, Google explained in its Statement of Work detailing "the responsibilities of Noser and the Project Services to be provided by Noser" that Google was "interested in *compatibility* with J2SE 1.5 . . . ." TX 2765 at 9, 12 (emphasis added).

B. Under *Sega*, elements that are functionally required for compatibility are not copyrightable, regardless of how they are used.

In *Sega*, the Ninth Circuit held that Accolade's copying and disassembly of Sega's firmware code was a fair use, because Accolade's purpose in copying was "for studying or examining the *unprotected aspects* of a copyrighted computer program . . . ." 977 F.2d at 1520 (emphasis added). Those unprotected aspects were "the functional requirements for compatibility with the Genesis console—aspects of Sega's programs that *are not protected by copyright.* 17 U.S.C. § 102(b)." *Id.* at 1522 (emphasis added).

The logical order of the Ninth Circuit's reasoning is important. The Ninth Circuit did *not* hold that the fair use doctrine allowed Accolade to copy aspects of Sega's programs that were required for compatibility. Instead, the Ninth Circuit held that functional requirements for compatibility are not protected by the Copyright Act in the first instance. That is, Accolade did not need to rely on the fair use doctrine to establish that it was entitled to copy functional

requirements for compatibility. Instead, it relied on the fair use doctrine to establish that it was allowed Accolade to copy and disassemble *all* of Sega's code to the extent necessary to determine what was functionally required for compatibility.

Because aspects of a computer program that are functionally required for compatibility are not copyrightable, it *does not matter what the defendant does with them.* Even if the defendant's product *is not compatible* with the plaintiff's product, the plaintiff still cannot assert infringement based only on the copying of unprotected elements. "The protection established by the Copyright Act for original works of authorship does not extend to the ideas underlying a work *or to the functional or factual aspects of the work.*" *Sega*, 977 F.2d at 1524 (emphasis added) (citing 17 U.S.C. § 102(b)). A "party claiming infringement may place *no* reliance upon any similarity in expression resulting from unprotectable elements." *Apple Computer, Inc. v. Microsoft Corp.*, 35 F.3d 1435, 1446 (9th Cir. 1994) (quotation marks and citation omitted; emphasis in original). Thus, because the SSO of the 37 API packages is functionally required for compatibility, Google was entitled to use that SSO, regardless of whether its use made Android fully compatible or not.

However, as noted, by implementing the SSO of the 37 API packages, Google intended to promote interoperability. And by implementing that SSO, Google increased the extent to which source code written for one platform will operate on the other.

Dated: May 23, 2012

KEKER & VAN NEST LLP

/s/ Robert A. Van Nest
By: ROBERT A. VAN NEST

Attorneys for Defendant
GOOGLE INC.